Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models

Tony Nowatzki Vinay Gangadhar Karthikeyan Sankaralingam
University of Wisconsin - Madison
{tjn,vinay,karu}@cs.wisc.edu

Abstract

General purpose processors (GPPs), from small inorder designs to many-issue out-of-order, incur large power overheads which must be addressed for future technology generations. Major sources of overhead include structures which dynamically extract the data-dependence graph or maintain precise state. Considering irregular workloads, current specialization approaches either heavily curtail performance, or provide simply too little benefit. Interestingly, well known explicit-dataflow *architectures eliminate these overheads by directly executing the data-dependence graph and eschewing instruction-precise recoverability. However, even after decades of research, dataflow architectures have yet to come into prominence as a solution. We attribute this to a lack of effective control speculation and the latency overhead of explicit communication, which is crippling for certain codes.*

This paper makes the observation that if both out-of-order and explicit-dataflow were available in one processor, many types of GPP cores can benefit from dynamically switching during certain phases of an application's lifetime. Analysis reveals that an ideal explicit-dataflow *engine could be profitable for more than half of instructions, providing significant performance and energy improvements. The challenge is to achieve these benefits without introducing excess hardware complexity. To this end, we propose the Specialization Engine for Explicit-Dataflow (SEED). Integrated with an inorder core, we see 1.67× performance and 1.65× energy benefits, with an Out-Of-Order (OOO) dual-issue core we see 1.33× and 1.70×, and with a quad-issue OOO, 1.14× and 1.54×.*

1. Introduction

As transistor scaling trends continue to worsen, causing severe power limitations, improving the performance and energy efficiency of general purpose processors has become ever more challenging. Great strides have been made in targeting *regular* codes, through the development of SIMD, GPUs, and other designs [6, 12, 19, 21, 30]. However, codes that are irregular, either in terms of control or memory, still remain problematic. Control irregularity includes divergent or unpredictable branches, and memory irregularity includes non-contiguous or indirect access[1].

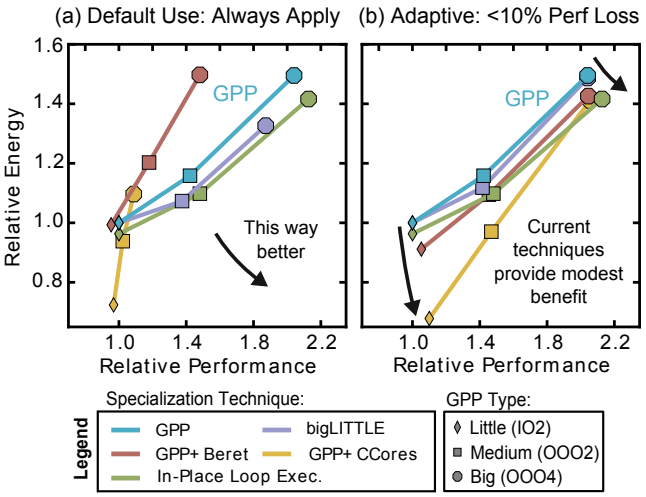


Figure 1: Energy/Perf. Tradeoffs of Related Techniques
(See Section 6 for methodology)

Existing Specialization Approaches Primarily, irregular codes are executed on general purpose processors (GPPs), which incur considerable overheads in per-instruction processing, both in extracting instruction-level parallelism and for maintaining instruction-precise state. Two broad specialization approaches have arisen to address these challenges. The first is to use simplified and serialized low-power hardware in commonly used low-ILP code regions for better energy efficiency. Examples include architectures like bigLITTLE [13] and Composite Cores [23], which switch to an inorder core when ILP is unavailable, and "accelerators" like BERET [14], Conservation Cores [35] and QsCores [36]. The other approach is to enhance the GPP for energy-efficiency, like adding μop caches, loop caches, and in-place loop execution techniques like Revolver [15].

To highlight the benefits and limitations of existing approaches targeting irregular codes, Figure 1(a) shows their energy and performance advantages when integrated into several GPP cores[2]. Figure 1(b) is similar, but here an oracle scheduler

[1]This paper examines irregular workloads by restricting to SPECint/Mediabench. Observations here apply to irregular workloads, unless specified.

[2]Note here that we allow switching arbitrarily at a fine-granularity and hence bigLITTLE subsumes Composite Cores.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA

ACM 978-1-4503-3402-0/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750380

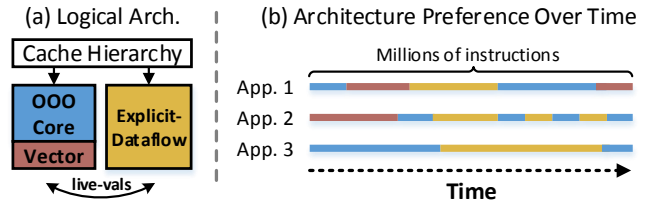
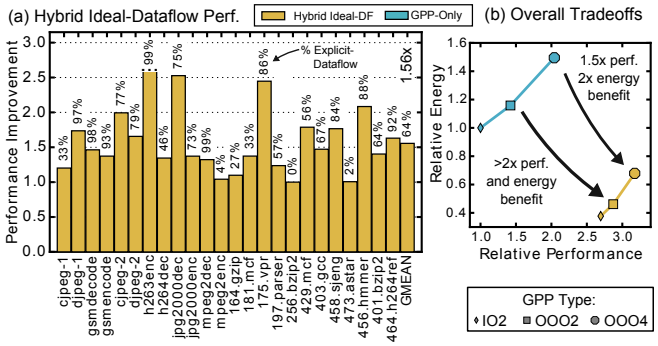


Figure 2: Taking Advantage of Dynamic Program Behavior



An "ideal" dataflow processor is only constrained by the program's control and data-dependencies, and not by any execution resources. It is also non-speculative, and incurs latency when transferring values between control regions. For its energy model, only functional units and caches are considered.

Figure 3: Potential of Ideal Explicit-Dataflow Specialization

only allows regions with slowdown of $< 10\%$. These results show that low-power hardware approaches are effective when integrated to small inorder cores ($1.5\times$ energy-efficiency), but usually cost too much performance to be useful for OOO GPPs. Techniques like in-place loop execution are also beneficial, but can only improve performance/energy by a few percent, because they rely on expensive instruction window, reorder-buffer and large multi-ported register-file access, even during loop specialization mode. Overall, speedup and energy benefits are limited to less than $1.1\times$ on large GPPs.

Dataflow The common feature of the above architectures is that they are fundamentally Von Neumann or "control flow" machines. However, there exist well-known architectures which eschew complex OOO hardware structures, yet can extract significant ILP, called *explicit-dataflow* architectures. These include early Tagged Token Dataflow [1], as well as the more recent TRIPS [5], WaveScalar [33] and Tartan [24]. But explicit-dataflow architectures show no signs of replacing conventional GPPs, for at least three reasons. First, control speculation is limited by the difficultly of implementing dataflow-based squashing. Second, the latency cost of explicit data communication can be prohibitive [3]. Third, compilation challenges have proven hard to surmount [9].

Overall, dataflow machines researched and implemented thus far have failed to provide higher instruction-level parallelism, and their theoretical promise of low power and yet high performance remains unrealized for irregular codes.

Unexplored Opportunity What has thus far remained unexplored is fine-grained interleaving of explicit-dataflow with Von Neumann execution – i.e. the theoretical and practical

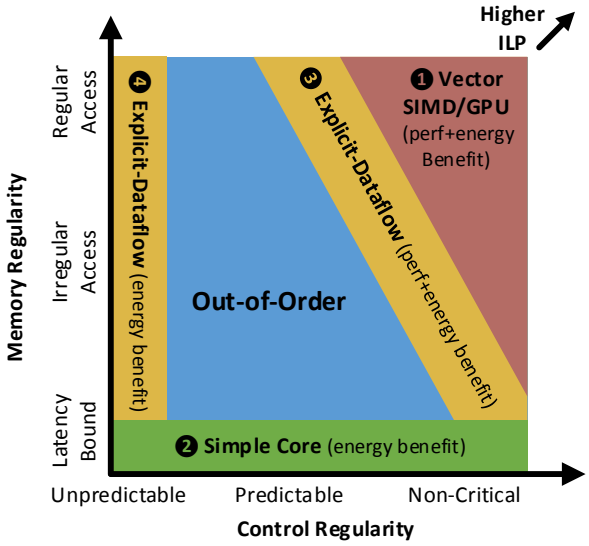


Figure 4: Arch. Effectiveness based on App. Characteristics

limits of being able to switch with low cost between an explicit-dataflow hardware/ISA and a Von Neumann ISA. Figure 2(a) shows a logical view of such a hybrid architecture, and Figure 2(b) shows the capability of this architecture to exploit fine-grain (several thousands to millions of instructions) application phases.

The potential benefits of an *ideal* hybrid architecture (ideal dataflow + four-wide OOO) are shown in Figure 3(a). Above each bar is the percentage of execution time in dataflow mode. Figure 3(b) shows the overall energy and performance trends for three different GPPs.

These results indicate that hybrid dataflow has significant potential, up to $1.5\times$ performance for an OOO4 GPP ($2\times$ for OOO2), as well as over $2\times$ average energy-efficiency improvement, significantly higher than previous specialization techniques. Furthermore, the preference for explicit-dataflow is frequent, covering around 65% of execution time, but also intermittent and application-phase dependent. The percentage of execution time in dataflow mode varies greatly, often between 20% to 80%, suggesting that phase types can exist at a fine grain inside an application.

When/Why Explicit-Dataflow? To understand when and why explicit-dataflow can provide benefits, we consider the program space along two dimensions: control regularity and memory regularity. Figure 4 shows our view on how different programs in this space can be executed by other architectures more efficiently than with an OOO core. Naturally, vector-architectures are the most effective when memory access and control is highly regular (see ❶). When memory latency bound (see ❷), little ILP will be available, and the *simplest possible* hardware will be the best (a low-power engine like BERET [14] or CCores [35]). An explicit-dataflow engine could also fill this role.

There are two remaining regions where explicit-dataflow has advantages over OOO. First, when the OOO processor's

issue width and instruction window size limits the achievable ILP (see ❸), explicit-dataflow processors can exploit this through more efficient hardware mechanisms, achieving higher performance and energy efficiency. Second, when control is not predictable, which would serialize the execution of the OOO core (see ❹), explicit-dataflow can execute the same code with higher energy efficiency.

Overall, this suggests that a heterogeneous Von Neumann/explicit-dataflow architecture with fine-granularity switching can provide significant performance improvements along with power reduction, and thus lower energy.

This Paper's Aims In this work, we explore the idea of extending the micro-architecture and integrating a new ISA for a simple dataflow-engine, where switching between the conventional and explicit-dataflow architecture can take advantage of changing program behavior. With such a hardware organization, many open questions arise – Are the benefits of fine-grained interleaving of execution models significant enough? How might one build a practical and small footprint dataflow engine capable of serving as an offload engine? Which types of GPP cores can get substantial benefits? Why are certain program region-types suitable for explicit-dataflow execution? We attempt to answer these questions through the following contributions:

- **Fine-grain dataflow opportunity** – Recognizing and quantifying the potential benefits from switching between OOO and explicit-dataflow at a fine grain.
- **Design of the Specialization Engine for Explicit-Dataflow (SEED)** – A design which exploits nested loops for simplicity, and yet is still widely applicable, and combines known dataflow-architecture techniques for high energy efficiency and performance. We also propose and evaluate essential compiler mechanisms.
- **Design-space exploration across GPP cores** – Exploring the benefits of dataflow heterogeneity by integrating SEED into little (Inorder), medium (OOO2) and big (OOO4) cores. We show all design points achieve $> 1.5\times$ energy benefit by power-gating the OOO core when SEED is active. For speedup, little, medium and big cores achieve $1.67\times$, $1.33\times$ and $1.14\times$ over the non-specialized design.
- **Connecting workload properties to dataflow profitability** – Showing that code with high memory parallelism, instruction parallelism, and branch unpredictability is highly profitable for dataflow execution.

Paper Organization We first develop the primitives required for fine-grain explicit-dataflow specialization (§2), then describe SEED's micro-architecture (§3) and compiler mechanisms (§4). We then discuss related work (§5). To understand the potential benefits, we present methodology (§6) and perform detailed evaluation and design space exploration (§7). We conclude by quantitatively discussing the implications of dataflow heterogeneity on future designs(§8).

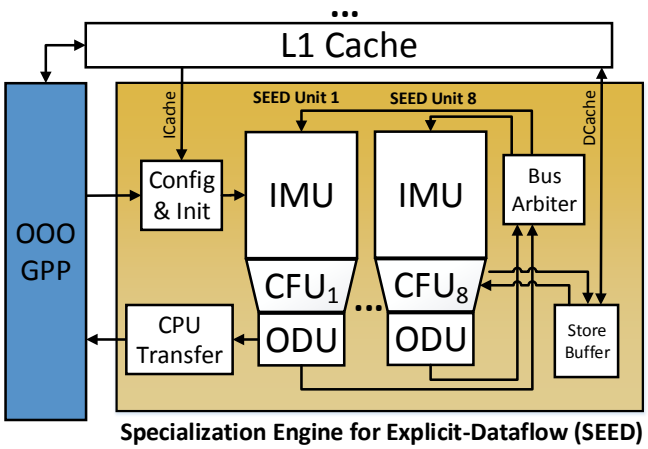


Figure 5: High-Level SEED Integration & Organization
(IMU: Instruction Management Unit; CFU: Compound Functional Unit; ODU: Output Distribution Unit)

2. SEED: An Architecture for Fine-Grain Dataflow Specialization

Our primary observation is the potential for exploiting the heterogeneity of execution models between Von Neumann and Dataflow at a fine grain. Attempting to exploit this raises this paper's main concern: *how can we exploit dataflow specialization with simple, efficient hardware?* We argue that any solution requires three properties: ① Has low-area and low-power, so that integration with the GPP is feasible. ② Is general enough to target a wide variety of workloads. ③ Achieves the benefits of dataflow execution with few overheads. Our codesign approach involves exploiting properties of frequently executed program regions, a combination of power-efficient hardware structures, and a set of compiler techniques.

First, we propose that requirement ①, low area and power, can be addressed by focusing on a common, yet simplifying case: *fully-inlined nested loops* with a limited total static instruction count. Limiting the number of per-region static instructions limits the size of the dataflow tags, and eliminates the need for an instruction cache; both of which reduce hardware complexity. In addition, ignoring recursive regions and only allowing in-flight instructions from a single context eliminates the need for tag matching hardware – direct communication can be used instead. Targeting nested-loops also satisfies requirement ②: these regions can cover a majority of real applications' dynamic instructions.

To achieve low-overhead dataflow execution, requirement ③, the cost of communication must be lowered as much as possible. We achieve this through a judicious set of micro-architectural features. First, we use a *distributed-issue* architecture, which enables high instruction throughput with *low ported RAM* structures. Second, we use a *multi-bus network* for sustaining instruction communication throughput at low latency. Third, we use *compound instructions* to reduce the data communication overhead.

Using the above insights creates new compiler requirements:

Arch. Parameter	**TRIPS** [31]	**Wavescalar** [33]	**CCA** [7]	**DySER** [11]	**BERET** [14]	**SEED**
Execution Units	Homogeneous FUs	Heterog. FUs and tag matching ①	Triangular mesh + heterog. FUs ②	Grid of heterogeneous FUs	Serialized compound FUs ③	Compound Functional Units (CFUs)
Storage Structures	Multiple SRAMS per Grid ①	Banked queues	Pipelined FIFOs ②	Pipelined FIFOs ②	CRAM, Internal Reg. Files	Single ported SRAM structures
Interconnection Network	Large 2D mesh network ①	Large hierarchical interconnect ①	Multi-level bus	Switches	Bus-based	Bus-based + Arbiter
Communication	Dynamic routing ①	Results, tokens across clusters ③	Configuration, results over bus	Tightly coupled with GPP ③	Config. messages, results	Fixed sized packet based
Control Flow Strategy	Dataflow predication ③	ϕ and ϕ_{-1} instructions	Control flow assertion	Predication-only ②	Speculates hot traces only ②	Switch instructions

Table 1: Suitability of Related Architectures for Explicit-Dataflow Specialization

①: Low-area & low-power; ②: Application Generality; ③: Low-overhead Dataflow;

1. Ability to create appropriately sized inlined nested loops matching the hardware constraints. 2. Algorithms for creating compound instruction groupings which minimize the communication overhead. For the first requirement, we can use aggressive inlining and loop nest analysis, and the second by employing integer linear programming models.

To address the architecture and compiler challenges, we propose SEED: Specialization Engine for Explicit-Dataflow, shown at a high level in Figure 5. Below, we overview the execution model and integration.

2.1. Execution Model and Core Integration

Adaptive Execution The model we use for adaptively applying explicit-dataflow specialization is similar to bigLITTLE, except that we restrict the entry points of acceleratable regions to fully-inlined loops or nested loops. This allows us to target custom hardware with a different ISA, using statically generated instructions. Targeting longer nested-loop regions also means a reduced overall cost of configuration and GPP core synchronization.

GPP Integration We integrate the SEED hardware with the same cache hierarchy as the GPP, as shown in Figure 5. This approach facilitates fast switching (no data-copying through memory), maintains cache coherence, and also eliminates the area of scratchpad memories and the associated need for programmer intervention. SEED also adds architectural state, which must be maintained at context switches. Lastly, functional units (FUs) could be shared with the GPP to save area (by adding bypass paths); this work considers standalone FUs.

Dataflow Style Similar to dataflow architectures like WaveScalar [33], control dependencies in the original program become data dependencies. The control flow is implemented by forwarding values to the appropriate location depending on the branch outcomes. Section 3 shows a more detailed dataflow example of a SEED program.

3. SEED Architecture

We begin the description of SEED by giving insight into why some architectural innovation is required and how our solution borrows mechanisms from related techniques. We then give an example SEED program which elaborates the basic mechanics of SEED execution. Subsequently, we detail the SEED microarchitecture from the bottom-up by describing SEED's submodules, its interconnection network and GPP integration.

We emphasize here that the organization of SEED is not the primary contribution of this paper, rather, it is a tool for understanding the potential of dataflow specialization.

3.1. Architectural Innovation

In exploring the opportunity of fine-grain explicit-dataflow, it is important to consider whether existing architectures would be sufficient. We list five related architectures in Table 1, and describe their execution and storage units and their strategy for value-communication and control flow. Each cell also lists our opinion of whether the design choice would not meet the previously discussed requirements (low-area/power, generality, and low-overhead dataflow).

TRIPS and Wavescalar are designed for whole-program dataflow execution, and use higher-power, higher-area structures. TRIPS uses a large dynamically routed mesh network and Wavescalar uses complex tag-matching and a large hierarchical interconnect. The remaining architectures have much lower power and area, but are not general enough. None of them can offload entire loop regions in general – only the computation in CCA and DySER or hot loop-traces in BERET.

However, aspects of these architectures can be borrowed: the principle of offloading to a dataflow processor from DySER, the concept of efficient compound FUs from BERET and mechanisms for efficient and general dataflow-based control from Wavescalar. The next section describes how SEED combines these design aspects using an example program.

3.2. Example SEED Dataflow Program

Figure 6 shows an example loop for a simple linked-list traversal, where a conditional computation is performed at each node. This figure shows the original program, control flow graph (CFG), and the SEED program. The SEED representation strongly resembles those of previous dataflow architectures, where the primary difference is that instructions are grouped here into subgraphs. Familiar readers may skip ahead.

Data-Dependence Similar to other dataflow representations, SEED programs follow the dataflow firing rule: instructions

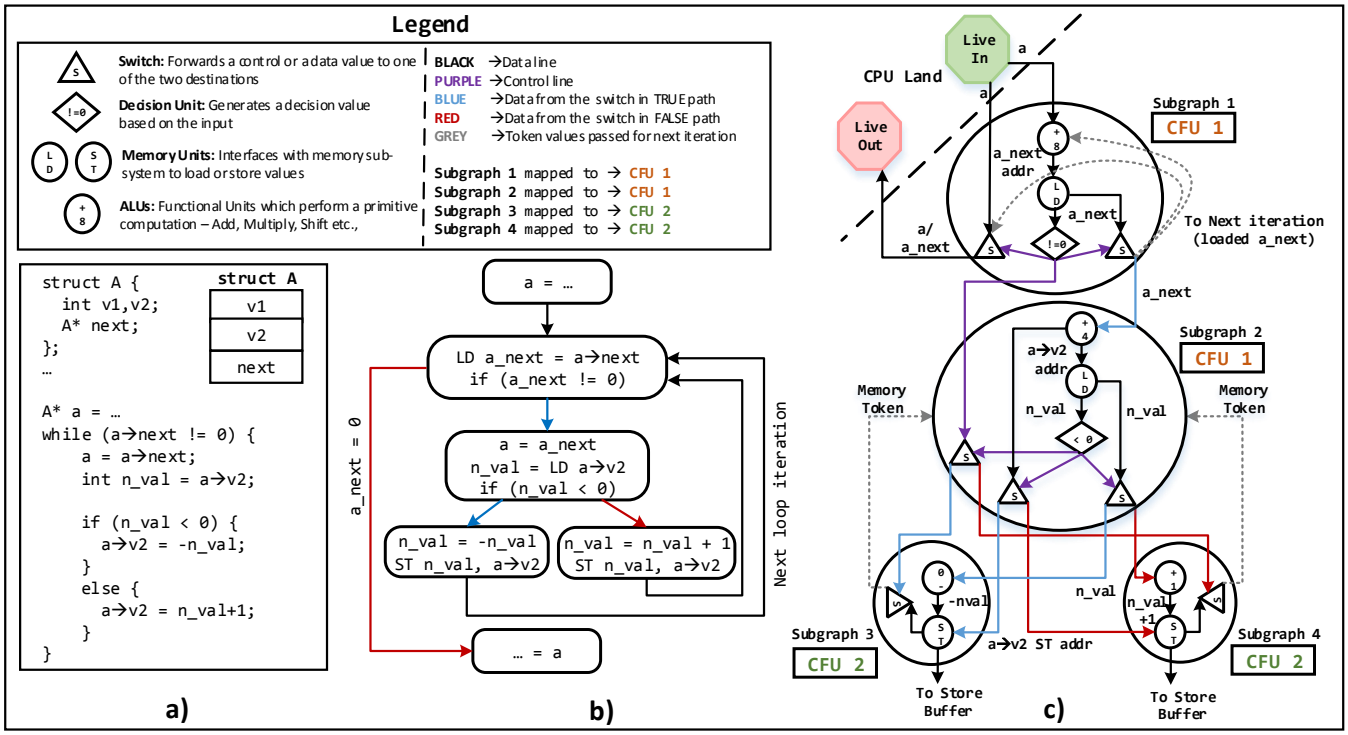


Figure 6: a) Example C loop; b) Control Flow Graph (CFG); c) SEED Program Representation;

execute when their operands are ready. To initiate computation, live-in values are sent. During dataflow execution, each instruction forwards its outputs to dependent instructions, either in the same iteration (solid line in Figure 6(c)), or in a subsequent iteration (dotted line). For example, the `a_next` value loaded from memory is passed on to the next iteration for address computation.

Control-Flow Strategy Control dependencies between instructions are converted into data dependencies. SEED uses a *switch* instruction, similar to other proposals, which forwards the control or data values to one of two possible destinations, depending on the input control signal. In the example, depending on the `n_val` comparison, `v2` is forwarded to either the `if` or `else` branch. This strategy enables control-equivalent regions to spawn simultaneously.

Enforcing Memory-Ordering SEED uses a software approach to enforce correct memory-ordering semantics. When the compiler identifies dependent (or aliasing) instructions, the program must serialize these memory instructions through explicit tokens. In this example, the stores of `n_val` can conflict with the load from the next iteration (e.g. when the linked list contains a loop), and therefore, memory dependence edges are required between these instructions.

Executing Compound Instructions To mitigate communication overheads, the compiler groups primitive instructions (e.g. adds, shifts, switches, etc.) into subgraphs and executes them on compound functional units (CFUs). These are logically executed atomically. The example program contains four subgraphs, mapped to two CFUs.

3.3. SEED Micro-Architecture

Our micro-architecture achieves high instruction parallelism and simplicity by using distributed computation units. The overall design is composed of 8 *SEED units*, where each SEED unit is organized around one CFU. The SEED units communicate over a network, as shown in Figure 5. We describe SEED unit internals and interconnect below (shown in Figure 7).

Compound Functional Unit (CFU) As mentioned previously, CFUs are composed of a fixed network of primitive FUs (adders, multipliers, logical units, switch units etc.), where unused portions of the CFU are bypassed when not in use. Long latency instructions (e.g. loads) can be buffered, and passed by subsequent instructions. An example CFU is shown in Figure 7 (c). Our design uses the CFU mix from existing work [14], where CFUs contain 2-5 operations. Our current design embeds integer hardware, but floating point (FP) units can be added either by instantiating new hardware or by adding bypass paths into the host processor's FP SIMD units.

CFUs which have memory units will issue load and store requests to the host's memory management unit, which is still active while using SEED. Load requests access a 32-entry store buffer for store-to-load forwarding.

Instruction Management Unit (IMU) The IMU, shown in Figure 7 (b), has three responsibilities:

1. **Storing instructions, operands & destinations:** The IMU has storage locations for 32 compound instructions, each with a maximum of four operands each, and we keep operand storage space for four concurrent loop iterations. This results in storage of 2600 bytes of data. All IMUs in

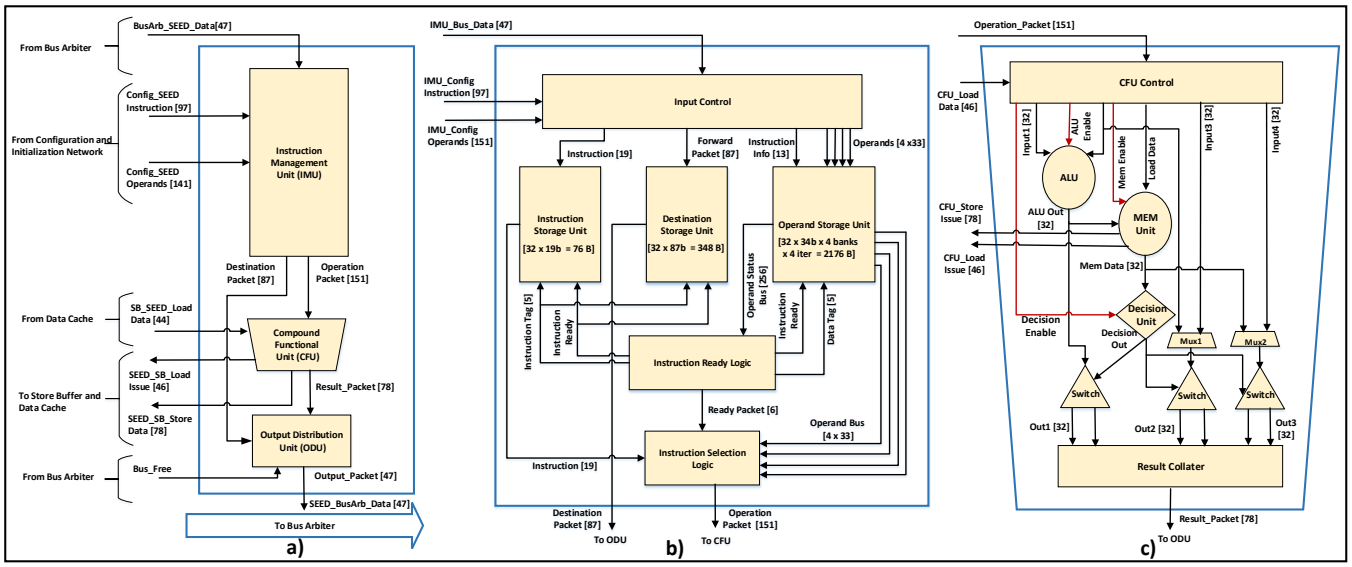


Figure 7: a) SEED Unit; b) IMU Micro-Architecture; c) CFU Micro-Architecture;

eight SEED units combined has ∼20KB of storage. The static instruction storage is roughly equivalent to a maximum of 1024 non-compound instructions.

2. **Firing instructions:** Ready logic monitors the operand storage unit, and picks a ready instruction (when all operands are available), with priority to the oldest instruction. Then the compound instruction and its operands and destinations are sent to the CFU.
3. **Directing incoming values:** The input control pulls values from the network to appropriate storage locations based on the incoming instruction tag.

The primary unique feature of the IMU is that it allows "unrolled" operand storage for four iterations of the loop. This allows instructions to directly communicate to dependent instructions without using power hungry tag-matching CAM structures at each execution node.

Output Distribution Unit (ODU) The ODU is responsible for distributing the output values and destination packets (SEED unit + instruction location + iteration offset), to the bus network, and buffer them during bus conflicts.

Bus Architecture and Arbiter SEED uses a bus interconnect for forwarding the output packets from the ODU to a data dependent compound instruction, present in either the same or another SEED unit. Note that this means dependent instructions communicating over the bus cannot execute in back-to-back cycles. To handle network congestion, the bus arbiter monitors the packet requests, and forwards up to three values on three parallel buses.

3.4. Integration With Core

The host core communicates with SEED to initialize configuration, send and receive input/output live values, and also during context switching. We discuss these next.

Interface with Host Code When inserting SEED regions into the GPP code, it is useful to keep both versions (e.g. in case the SEED region does not last long enough to warrant offloading). Therefore, two instructions are inserted into the original code. The first, `SEED_CONFIG`, is inserted at the earliest dominating basic block in the host code, which signals the SEED unit to begin the *Configuration Stage*. This instruction contains the relevant memory addresses for configuration bits. The second added instruction, `SEED_BEGIN`, is a type of conditional branch, which transfers control to SEED if it is predicted to run for long enough to mitigate overheads. This instruction signals the live value transfer from GPP registers.

Configuration Stage This stage populates the instruction and destination storage units for a particular region, as well as initialize any loop-invariant constants in the operand storage unit. Configuration data is read through the instruction cache, and streamed to each IMU. After configuring storage, the "live-in" packets can be sent to initiate dataflow computation.

Context Switching To handle context switching, the current live operands must become part of the architectural state. To mitigate the overhead, we delay context switches until the current inner-loop iterations quiesce. In the worst case, we estimate needing to save 2KB of data, though typically the amount of live data is much less.

Region-Lifetime Prediction It may be unknown how long a region lasts at compile time, and the overheads of switching to SEED may be too high if the duration is too short. Also, if the region lasts long enough, it will be worthwhile to power-down the non-stateful parts of the OOO core. To enable these decisions, we keep a simple direct-mapped table of the running average times of different SEED regions. If it is predicted to be short, we do not enter the SEED region, and if long, the `SEED_BEGIN` will initiate the power-gating on non-stateful parts of the GPP core.

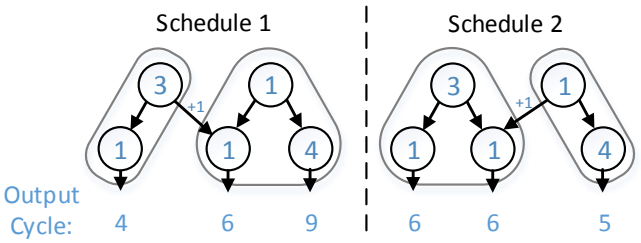


Figure 8: CFU Scheduling Example. Each operation is labeled with its latency, and compound insts. are circled.

4. SEED Compiler Design

The two main responsibilities of the compiler are determining which regions to specialize and scheduling instructions into CFUs inside SEED regions, which we describe in this section.

4.1. Region Selection

Finding Suitable Regions As SEED may only execute fully-inlined nested-loops, the compiler must find or create such regions. There are two main goals: 1. Finding small enough regions to fit the hardware, and 2. Not hurting performance by aggressively applying SEED, when either the OOO core (through control speculation) or the SIMD units would have performed better.

For the first goal, a bottom up traversal of the loop-nest tree can be used to find appropriately sized regions. Enough space can be left for unrolling inner loops.

For the second goal, either static or dynamic options are possible. For the static approach, simple heuristics will likely suffice – i.e. do not perform explicit-dataflow when control is likely to be on the critical path. A dynamic approach can be more flexible; for example, training on-line predictors to give a runtime performance estimate based on per-region statistics. Other works have shown such mechanisms to be highly effective [23, 28], and we therefore do not evaluate or implement this aspect of the compiler/runtime. Instead, we use an oracle scheduler, as described in the evaluation.

4.2. Instruction Scheduling

The goal of the instruction scheduler is to form compound instructions such that expected execution time and energy cost is minimized. The scheduler can affect performance in two ways, either by minimizing communication cost or by minimizing the expected critical path latency.

Figure 8 shows the importance of an effective scheduler with two example schedules of the same region. Note here that compound instructions can only begin execution once all inputs arrive, and that each inter-CFU communication costs an additional cycle. This example shows that small differences in the schedule can effect the critical path by many cycles.

To solve this problem, we use a declarative approach by utilizing an integer linear programming formulation. We specifically extend a general scheduling framework for spatial architectures [27] with the ability to model instruction bundling. We formally describe this next and present brief results.

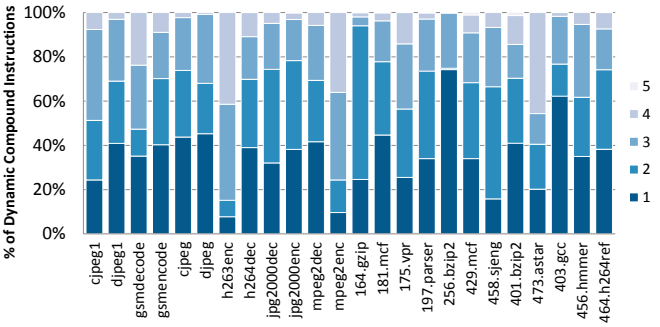


Figure 9: Compound Instruction Size Histogram

Formulation of Integer Linear Program Consider a computational graph made of vertices $v \in V$, where $G_{v_1 v_2}$ represents data dependencies between vertices v_1 and v_2. Similarly, consider a hardware graph of computational resources $n \in N$, where $H_{n_1 n_2}$ represents the connections of the hardware substrate between n_1 and n_2. The compatibility between hardware resources and computation vertices is given by the set C_{vn}. The goal is to attain a mapping M_{vn} from computation vertices to hardware resources, such that the expected latency is minimized. The formulation[3] is in Table 2, and variables and parameters are summarized in Table 3.

Evaluation of CFU Scheduler Figure 9 is a histogram of per-benchmark compound instruction sizes, showing on average 2-3 instructions. This is relatively high considering that compound instructions cannot cross control regions. Some singletons are necessary, however, either because control regions lack dependent computation, or because combining certain instructions would create additional critical-path dependencies.

5. Related Work

Dataflow Architectures The notion of merging the benefits of Von Neumann with dataflow machines is far from new. A large body of work in the dataflow paradigm is in executing explicitly parallel programs, like TTDA does with the Id programming language [1]. In that context, Iannucci proposes a hybrid architecture which adds a program counter to the TTDA to execute explicitly parallel programs more efficiently [18]. Along opposite lines, Buehrer and Ekanadham introduce another hybrid architecture which introduces mechanisms to support both sequential and explicitly parallel programming languages [4] for ease of transitioning. These works are orthogonal to ours, as they are attempting to target different programming models.

That said, SEED derives significant inspiration from the previous decades of dataflow research. One example is the Monsoon architecture [29], which improves the efficiency of matching operands by using an Explicit Token Store. This essentially eliminates complex matching hardware by allocating memory frames for instruction tokens and using offsets into

[3]The formulation here omits some details from our implementation for jointly optimizing communication and load balancing.

Integer-Linear Equation	Explanation
$\forall_{v \in V} \sum_{n \in C_{vn}} M_{vn} = 1$ $\forall_{v \in V} \sum_{n \notin C_{vn}} M_{vn} = 0$	These equations enforce that all computational vertices are mapped to exactly one compatible node.
$\bigvee_{\substack{v_1 v_2 \in G \\ n \in N}} B_{v_1 v_2} \geq M_{v_2 n_2} - \sum_{\substack{n_1 \in C_{v_1 n_1} \\ n_1 n_2 \in H}} M_{v_1 n_1}$	This constraint enforces that either a vertex's inputs are either directly routed through a hardware input, or they are executed on separate instances of a CFU (indicated by $B_{v_1 v_2} = 1$).
$\bigvee_{v_1, v_2, v_3 \in P} (1 - B_{v_1 v_2}) + (1 - B_{v_2 v_3}) - 1 \leq (1 - B_{v_1 v_3})$ $\forall_{v_1, v_2 \in G} \; B_{v_1 v_2} = B_{v_2 v_1}$	This enforces boundary transitivity: if there is no boundaries between v_1 and v_2, and v_2 and v_3, then there can not be a boundary between v_1 and v_3. This is only enforced if all nodes can possibly map together, indicated by P.
$\bigvee_{v_1, v_2, n \in C_{v_1 n} \cap C_{v_2 n} \cap P_{v_1 v_2}} M_{v_1 n} + M_{v_2 n} \leq B_{v_1 v_2} + 1$	This constraint makes sure that two nodes that could possibly map to each other are only allowed to map to the same hardware node if they are on the same CFU instance.
$\forall_{v_1, v_2 \in G} \; T_{v_2} \geq B_{v_1 v_2} + L_{v_1} + T_{v_2}$ $\bigvee_{v_1, v_2 \in P \cap v_i, v_1 \in G} T_{v_2} \geq (B_{v_i v_1} - B_{v_1 v_2} - 1) * M + L_{v_i} + T_{v_i}$ $\forall_v \; LAT \geq T_v$	These equations model timing. The first enforces simple timing, with an extra cycle if a boundary exists between CFUs. The second uses a Big-M formulation to enforce that the CFU instance cannot start until all inputs arrive. The third computes the final latency, which is the objective for minimization.

Table 2: Integer linear programming model for SEED Scheduling

Var/Param	Explanation
$G_{v_1 v_2}$	Graph of data dependencies.
L_v	Latency of operation v.
$P_{v_1 v_2}$	Describes whether v_1 could possibly be grouped with v_2 (computed off-line).
$H_{n_1 n_2}$	Connections between all hardware compound FUs.
C_{vn}	Compatibility between operation v and FU n.
$B_{v_1 v_2}$	Binary var. representing if v_1 and v_2 are mapped to different CFU instances (i.e. a *boundary* between them).
T_v	Variable for execution time of each operation.
M_{vn}	Binary var. representing operation to FU mapping.

Table 3: Scheduling Notation Summary

the frame for token locations. Our strategy of using explicit offsets into the operand buffers provides similar benefits.

Core Enhancements Revolver's [15] in-place loop execution somewhat resembles the in-place nested-loop acceleration of SEED, but uses higher-power structures. Another related OOO-enhancement is the ForwardFlow [10] architecture, which is also a CAM-free execution substrate using explicit pointer-based communication of values. Though it is more energy-efficient than a typical OOO design, it still suffers overheads of fetch and decode, centralized register-file access, and still must dynamically build the dependence graph.

Heterogeneity and Specialization Venkat et al. demonstrated that due to the characteristics of existing ISAs, there is potential efficiency gain through adaptive execution [34]. Our work exploits the same insight, but departs more radically from traditional Von Neumann ISAs.

Besides the accelerators described in the introduction, the most relevant architecture is XLOOPs [32], which is a recent design targeting inner loops with specific loop-dependence patterns. Though its high-level adaptive specialization paradigm is similar, along with some targeted code properties, the microarchitecture is vastly different, and will favor different codes. One benefit of SEED is its ability to target coarser grain regions, enabling more effective power-gating of the OOO core.

Suite	Benchmarks
Mediabench	cjpeg, djpeg, gsmdecode, gsmencode cjpeg2, djpeg2, h263enc, h264dec, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc
SPECint	164.gzip, 181.mcf, 175.vpr, 197.parser, 256.bzip2 429.mcf, 403.gcc, 458.sjeng, 473.astar, 456.hmmer

Table 4: Benchmarks

Another type of hybrid model is DySER [11], which accelerates the computation in an access-execute paradigm with an explicit-dataflow substrate. A full system FPGA-based evaluation shows that DySER is best suited to data-parallel workloads, where the cost of communication can be amortized with vectorization [16].

Finally, a concurrent work is the Memory Access Dataflow (MAD) architecture [17], which augments the GPP with a dataflow substrate for targeting memory access program phases. These phases occur either because the code is naturally memory intensive, or because an attached in-core accelerator offloads most of the computation. Conceptually, our work differs in that it explores the benefits of hybrid dataflow execution in both memory and computation intensive regions. In terms of the microarchitecture, they are both essentially speculation-free dataflow execution of nested-loops, and use queue-like structures for data storage. For computation, MAD uses a spatial grid of statically routed FUs, while SEED uses clustered-instruction execution.

6. Evaluation Methodology

Benchmark Selection The benchmarks we chose were from SPECint and Mediabench [20], representing a variety of control and memory irregularity, as well as some regular benchmarks (see Table 4).

GPP	Characteristics
Little (IO2)	Dual Issue, 1 load/store port.
Medium (OOO2)	64 entry ROB, 32 entry IW, LSQ: 16 ld/20 st, 1 load/store ports, speculative scheduling.
Big (OOO4)	168 entry ROB, 48 entry IW, LSQ: 64 ld/36 st, 2 load/store ports, speculative scheduling.

Table 5: GPP Cores

GPP Characteristics All cores are x86, have 256-bit SIMD, and have a common cache hierarchy: a 2-way 32KiB I$ and 64KiB L1D$, both with 4 cycle latencies, and an 8-way 2MB L2$ with a 22 cycle hit latency. Also, to exclude the effects of frequency scaling, all cores run at 2Ghz. The differences between GPP configurations are highlighted in Table 5. The OOO4 has 3 ALUs, 2FPs, and 1 Mul/Div unit, which are scaled according to the GPP issue width.

Modeling Specialization Architectures For studying the performance and energy of ideal dataflow and SEED architectures, as well as other specialization techniques like BERET, it is necessary to model the compiler. For this, we analyze a dynamic trace of instructions using critical path analysis [8], and augment it with a tool which reconstructs the program IR. We use this IR to perform instruction scheduling and other necessary compiler analysis. This tool is discussed in more detail here [26].

Simulator Implementation Our simulator consists of the above analysis and instruction translation tool integrated with gem5 [2]. McPAT 1.2 [22] and CACTI [25] models are used for power/energy of both accelerators and the GPP, and we use the 22nm configuration setting. For simulating the workloads, we fast-forward past the initialization regions, and use 200 million instruction traces.

7. Evaluating Dataflow-Specialization Potential

To understand the potentials and tradeoffs of dataflow specialization, while specifically exploiting nested-loop regions, this section attempts to answer the following questions:

Q1. Are nested loop regions common enough to be beneficial?
Q2. Is the proposed design practical: what is the area cost?
Q3. How much performance can targeted regions provide?
Q4. What are the sources of performance differences?
Q5. Which GPP cores can we enhance with our technique?
Q6. Would it still be useful if GPPs were more efficient?
Q7. Besides performance/energy, are there other benefits?
Q8. How does SEED compare to related approaches?

Q1. Are nested loop regions prevalent enough?

Here we explore the consequences of targeting fully-inlined nested loops, in terms of static size and region duration. For comparison, we study three program structures which are commonly used for specialization: loop traces (repeated basic-block traces), inner loops, and loop nests. Note that this study considers static binaries, and tradeoffs for dynamically-linked binaries will differ.

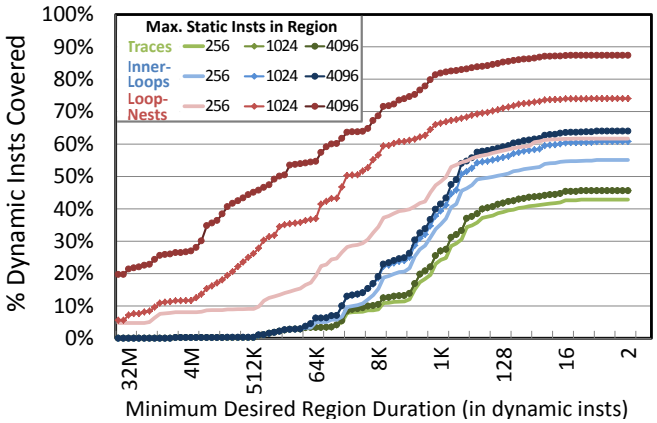


Figure 10: Cumulative % contribution for decreasing dynamic region lengths, shown for different static region sizes.

Module	Area (mm^2)	Module	Area (mm^2)
IMU	0.034	Internal Network	0.058
CFU	0.011	Total SEED Unit	0.114
ODU	0.010	Bus Arbiter	0.016
Total (8 SEED Units + Bus Arbiter)			0.926

Table 6: SEED Area Breakdown

Figure 10 shows cumulative distributions of dynamic instructions coverage by dynamic region granularity. For instance, considering regions with a duration of 8K dynamic instructions or longer (x-axis), nested loops can cover 70% of total instructions, while inner loops and traces can cover about 20% and 10% respectively. Considering *any* duration of region, the total instruction coverage is 88%, 64%, and 45% for nested loops, inner loops, and traces, respectively. Also, nested loops greatly increase the region duration (1K to 128K for 50% coverage).

For each region type, we also present different maximum region sizes: 256, 1024 and 4096 instructions. Targeting 1024 instruction regions (the effective size of SEED) presents a good tradeoff between total static instructions (more hardware cost) and dynamic region length.

Answer: *Yes, nested loops can target 24% more than inner loops, up to 88% of all instructions, and they allow longer duration regions.*

Q2. Is the proposed design practical?

To determine the area, we have implemented the SEED architecture in Verilog and synthesized the design using a 32nm standard cell library with the Synopsys Design Compiler. CACTI [25] was used for estimating SRAM area. Our results show that each SEED unit occupies reasonable area and all eight SEED units and bus arbiter together take up an area of 0.93 mm^2. Table 6 shows the area breakdown.

We also synthesized the design for 2 GHz, and the estimated power is 90mW based on its default activity factor assumptions

	Benchmark	Func. for SEED Region	% Exec. Insts	Vect-orized	OOO4 IPC	SEED IPC	Ideal-DF IPC	SEED En-Red.	BPKI	BMPKI	$MPKI	Explanation
Perf. > OOO4	jpg2000dec	jas_image_encode	50%		2.5	12.8	21.8	9.1	101	0	0	High Exploitable ILP
	429.mcf	primal_bea_mpp	37%		0.8	2.8	8.3	4.6	152	10	96	Higher Memory Parallelism
	cjpeg-1	encode_mcu_AC_refine	24%		2.5	5.9	6.2	4.2	48	0	2	Indirect Memory + High ILP
	181.mcf	primal_bea_mpp	31%		0.9	1.8	9.6	3.0	170	8	106	Higher Memory Parallelism
	djpeg-2	ycc_rgb_convert	33%		2.7	5.4	12.0	3.5	29	0	0	Indirect Memory + High ILP
	456.hmmer	Viterbi*	73%		2.9	5.4	7.3	4.5	32	0	4	High Exploitable ILP
	458.sjeng	std_eval	5%		2.4	3.7	4.1	3.8	126	5	0	High Exploitable ILP
	gsmdecode	Gsm_Short_Term_Syn...	61%		2.4	3.1	3.4	4.5	92	0	0	High Exploitable ILP
	cjpeg-2	compress_data	48%	✓	2.2	2.7	4.9	3.5	58	8	0	High Exploitable ILP
Perf. ≈ OOO4	gsmencode	Gsm_Long_Term_Pred...	49%		1.9	2.2	2.7	3.5	5	0	0	Modest ILP + Comm. Overheads
	djpeg-1	decompress_onepass	39%		2.6	2.7	3.6	3.6	18	1	0	Indirect Memory + Moderate ILP
	h263enc	MotionEstimation	98%		2.0	1.9	8.7	3.2	18	0	0	Comparable Performance
	164.gzip	inflate	23%		1.9	1.7	2.3	2.0	81	0	10	Modest ILP + Comm. Overheads
	473.astar	wayobj::fill	96%		1.1	1.0	1.1	3.3	114	31	2	Avoids Branch Misses, Modest ILP
	h264dec	decode_one_macroblock	21%		0.4	0.4	0.4	1.9	39	0	0	Comparable, Low ILP
	jpg2000enc	jpc_enc_encpkt	3%		2.1	1.8	2.0	1.3	135	6	2	Comparable Performance
Perf. < OOO4	403.gcc	ggc_mark_trees	4%		0.5	0.4	0.4	1.0	66	2	2	Comparable, Low ILP
	464.h264ref	SetupFastFullPelSearch	29%		1.5	1.3	1.7	2.7	40	0	0	Short Region (340 Dyn Insts)
	175.vpr	try_swap	49%		1.4	1.2	6.9	2.1	88	17	5	Avoids B-Misses, Comm. Overhead
	mpeg2enc	fullsearch.constprop.3	93%		1.9	1.5	2.9	3.9	17	0	0	Moderate ILP, Comm. Overhead
	mpeg2dec	conv422to444	31%		2.7	2.1	3.0	2.8	68	0	2	Moderate ILP, Comm. Overhead
	197.parser	restricted_expression	17%		3.3	1.6	3.7	1.4	108	0	0	Short Region (300 Dyn Insts)
	401.bzip2	BZ2_compressBlock	31%	✓	4.3	1.5	1.5	0.8	97	3	3	Region Vectorized
	256.bzip2	compressStream	99%	✓	13.5	2.0	2.0	0.4	83	0	0	Region Vectorized

Table 7: Region-Wise Comparison of OOO4 to SEED, Showing *only* top region per benchmark, Highest to Lowest Relative Perf. (%Exec. Insts: % of original program executed by SEED; Vectorized: whether the GPP vectorized the region, SEED IPC: Effective IPC of SEED, Ideal-DF IPC: IPC of Ideal-dataflow, En-Red: SEED's Energy Reduction, BPKI: Branches per 1000 μops, BMPKI: Branch Mispred. per 1000 μops, $MPKI: Cache misses per 1000 μops)

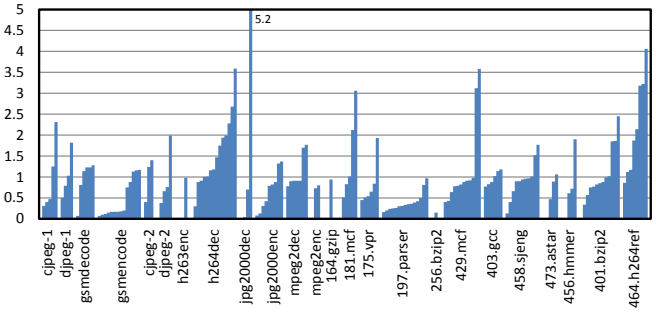


Figure 11: Per-Region SEED Speedups

for the datapath[4].

Answer: *The simple design and low area/power quantitative results show that the SEED unit is practical.*

Q3. How much performance benefit is possible?

To understand if there are potential performance benefits, we compare the speedups of SEED to our most aggressive design (OOO4) on the most frequent nested-loop regions of programs (each >1% total insts). The results, in Figure 11, show that different regions have vastly different performance characteristics, and some are favored heavily by one architecture.

Answer: *Nearly 3-5× speedup is possible, and many regions show significant speedup.*

[4]For fairness of comparing against McPAT-based GPP models, we have also used a McPAT-based model for SEED. For performance benefit regions (vs OOO4) the McPAT model reports an average power of 125mW, meaning this model should be conservative.

Q4. What are the sources of performance differences?

Table 7 presents details on the highest contributing region from each benchmark. Note that the SEED IPC is an *effective IPC* which uses the GPP's instructions as total instructions. This allows easier comparison, as many instructions for loading immediates and managing register spilling are not required in SEED. We discuss these in three categories:

Perf.&Energy Benefit Regions Compared to the OOO4-wide core, SEED can provide high speedups for certain applications, coming from the ability to exploit higher ILP in compute-intensive regions and from breaking the instruction window barrier for achieving higher memory parallelism.

In the first category are `jpg2000dec`, `cjpeg` and `djpeg`, which can exploit ILP past the issue width of the processor, while simultaneously saving energy by using less complex structures. Often, these regions have indirect memory access which precludes SIMD vectorization. In the second category are `181.mcf` and `429.mcf`, which experience very high cache miss rates, and clog the instruction window of the OOO processor. SEED is only limited by the store buffer size on these benchmarks.

Energy Benefit-Only Regions These regions have similar performance to the OOO4, but are more energy efficient by 2-3×. Overall, ILP tends to be lower, but control is mostly off the critical path, allowing dataflow to compete. This is the case for `djpeg-1` and `h264dec`. Benchmarks like `gsmencode` and `164.gzip` actually have some potential ILP advantages, but are burdened by communication overhead between SEED

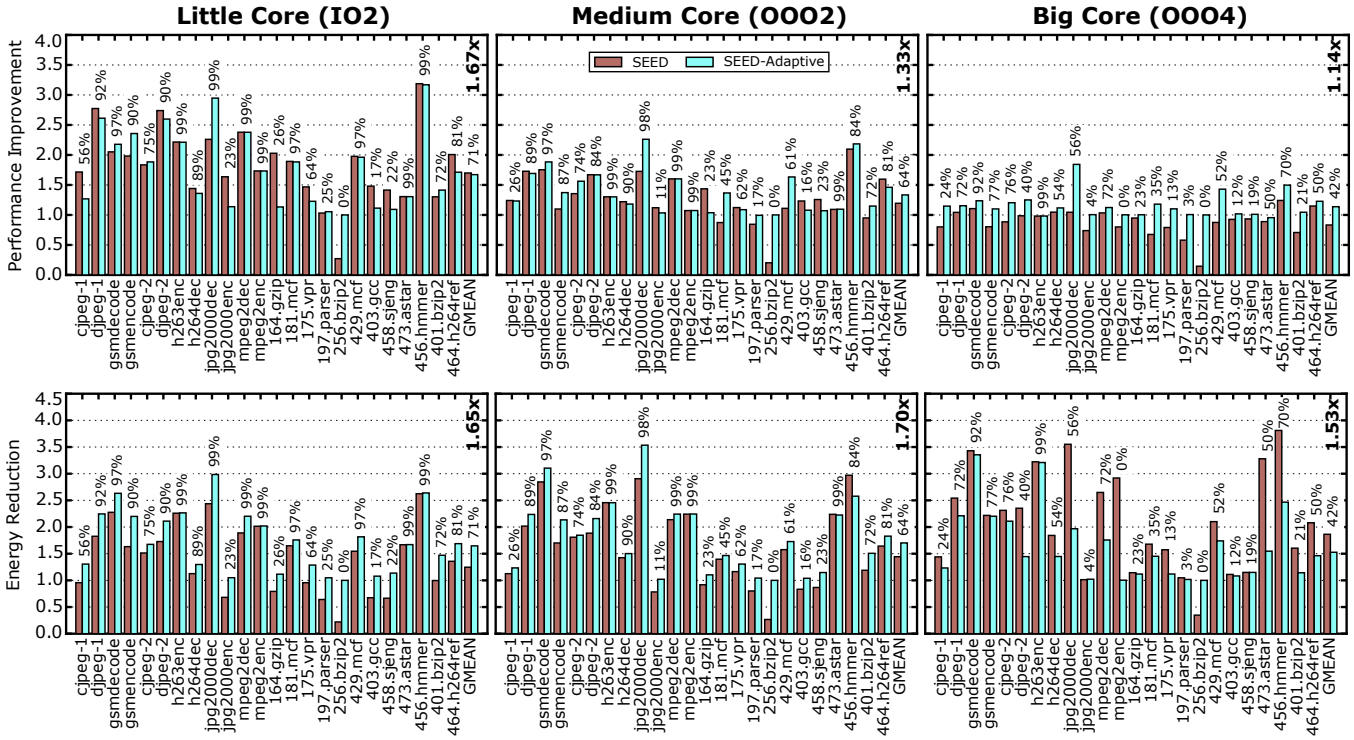


Figure 12: SEED Specialization for Little, Medium, and Big Cores

units. Benchmark `h263enc` actually has a very high potential ILP, but requires multiple *instances* of the inner loop (not just iterations) in parallel, which SEED does not support.

Contrastingly, benchmarks `473.astar` and `jpg2000enc` have significant control, but still perform close to the OOO core. These benchmarks make up for the lack of speculation by avoiding branch misses and relying on the control-equivalent spawning that dataflow provides.

Perf. Loss Regions Several SEED regions lose performance versus the OOO4 core, shown in the last set of rows in Table 7. The most common reason is additional communication latency on the critical path, affecting regions in `403.gcc`, `mpeg2dec` and `mpeg2enc`. Also, certain benchmarks have load-dependent control, like `401.bzip2`, causing a low potential performance for dataflow. These are fundamental dataflow limitations. In two cases, configuration overhead hurt the benefit of a short-duration region(`464.h264ref` and `197.parser`). In practice, these regions would not be executed on SEED. Finally, some of these regions are vectorized on the GPP, and SEED is not optimized to exploit data-parallelism. This affects `401.bzip2` and `256.bzip2`.

Answer: *Speedups come from exploiting higher memory parallelism and instruction parallelism, and avoiding misspeculation on unpredictable branches. Slowdowns come from the extra latency cost on more serialized computations.*

Q5. Which GPP cores can we enhance with SEED?

Here we consider integrating with a little, medium, and big core, as outlined in Table 5. To eliminate compiler/runtime heuristics on when to accelerate, we consider using an oracle scheduler, which uses perfect information to decide when to use the OOO core, SEED, or SIMD. We report results for performance and energy reduction of all cores in Figure 12. The first bar in each graph shows the relative metric to the baseline, when *always* using SEED. The second bar, "adaptive," shows the result of the oracle scheduler, optimizing for Energy-Delay product, and not allowing any regions which degrade performance by more than 10%. We discuss the implications for each GPP type below.

Little GPP (IO2) For the little core, SEED provides a geometric mean performance and energy improvement of about 1.65×, and SEED runs for 71% of the execution time. For these benchmarks, SEED mainly loses performance on vectorized workloads like `256.bzip2`.

Medium GPP (OOO2) For the medium core, SEED is the chosen execution substrate for 64% of the execution time, providing energy reduction of 1.7×, and performance of 1.33×. Even if always chosen, in only four cases does it hurt performance, and in most cases energy-efficiency gain is significant.

Big GPP (OOO4) For the big core, for reasons described in the previous section, SEED is chosen less, around 42% of execution time. Overall though, it still provides 1.14× performance and 1.53× energy efficiency improvement.

Answer: *All cores can achieve significant energy benefits; little and medium cores can achieve significant speedup; and big cores receive modest performance improvement.*

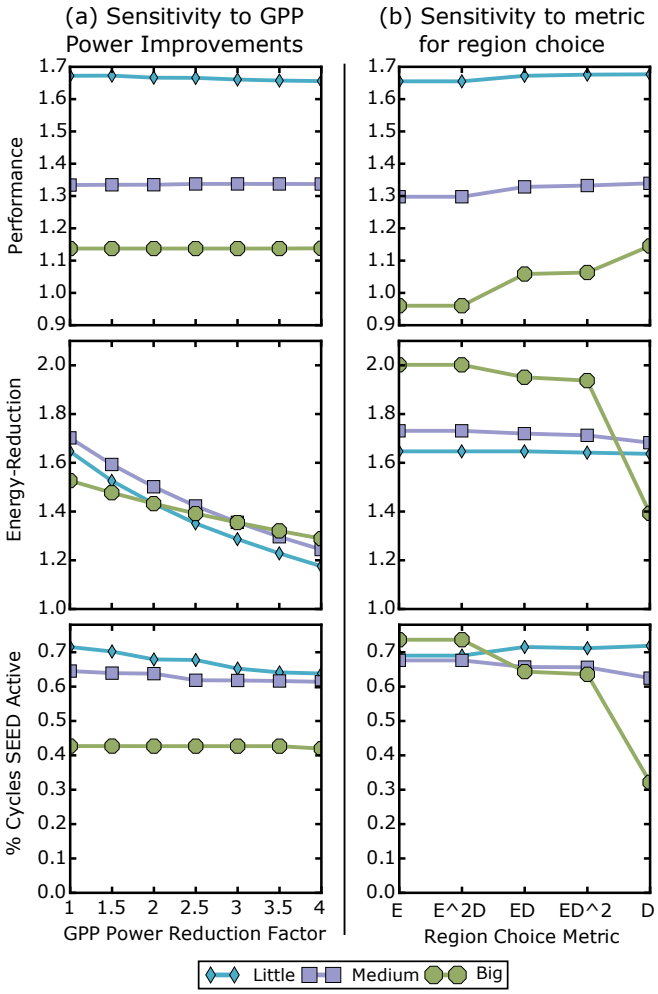


Figure 13: Sensitivity to GPP Improvements and Region Choice Metric

Q6. Would this still be useful if GPPs were more efficient?

Figure 13(a) shows the performance, energy, and percentage of cycles that SEED is active across all workloads, while reducing the power of all GPP structures (not including the SEED unit). The x-axis is the factor by which the GPP is made more power efficient (1 means no change).

Naturally, the energy-reduction of all GPPs decrease as a direct effect of the changing parameters. More interestingly, the percentage of time SEED is chosen drops only by a few percent (and only for the little and medium cores), even if the GPP becomes 4× more energy efficient.

Answer: *Even future generations of power-efficient GPPs could take advantage of explicit-dataflow specialization.*

Q7. Beside energy/speedup, are there other benefits?

Figure 13(b) shows the effects of varying the region choice metric from energy-efficiency (E), to energy-delay (ED), and up to performance (D). Naturally, the little and medium cores are not particularly sensitive to the region choice metric, which is intuitive because SEED is a faster, yet still primarily lower-power design. The big core is quite sensitive; by optimizing for energy-efficiency (and using SEED more often) it can trade off 20% performance for over 40% energy efficiency on average.

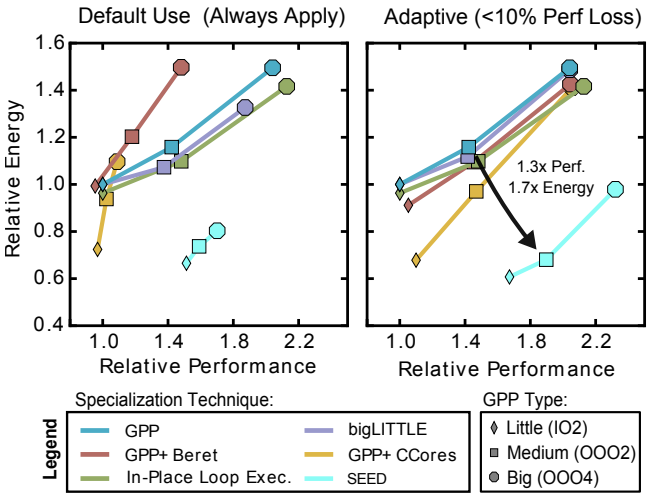


Figure 14: Comparison With Other Specialization Techniques

Answer: *Explicit-dataflow specialization provides a micro-architectural mechanism for trading-off performance and energy-efficiency on large cores.*

Q8. How does SEED compare with other specialization approaches?

An updated version of the first figure, Figure 14, compares SEED to existing techniques. In non-adaptive mode, SEED provides energy improvements for all cores, and performance enhancements for the inorder and medium cores. For adaptive mode, SEED improves both metrics across GPP cores types, significantly more than existing approaches. In terms of the overall design space, the OOO2+SEED cannot beat an OOO4 on average, but reaches within 15% performance (with much lower power). Also, though SEED improves OOO4 performance only modestly, the energy efficiency of OOO4+SEED is that of a simpler OOO2 GPP.

Answer: *Explicit-Dataflow specialization has significant potential beyond existing techniques across core types.*

8. Discussion and Conclusions

We conclude the paper by discussing how the potential of dataflow-specialization affects future core-design decisions.

Generalizing SEED's results to apply for the paradigm of fine-grain dataflow/Von Neumann execution, this paper's findings suggest two positions for what this means for future designs: A pessimistic view is that if very-high performance on irregular code is necessary, dataflow is not an alternative to building big OOOs. Even mobile devices are using "big" cores, and if we move to even larger cores, opportunities for improving performance with dataflow specialization become even less. A more optimistic view is that fine-grain dataflow presents a compelling opportunity for retaining the high per-

formance of medium to big cores, and lowers their energy by 30% to 40% across difficult-to-improve irregular applications.

Also, we have not yet achieved the benefits of ideal specialization, either in terms of application coverage or total speedup and energy gains. An open question is whether another roughly 30% of instructions in irregular workloads can be specialized if more flexible, yet still efficient, hardware mechanisms for explicit-dataflow specialization can be developed. Or alternatively, can a different set of data-parallel mechanisms be used to augment SEED to target more regular workloads, completely obviating the need for short-vector SIMD extensions.

While this paper's analysis and claims hinge upon the ability to add significant hardware and a new ISA and associated compiler, an interesting opportunity along another dimension is whether the same benefits can be achieved without (or with minimal) ISA modifications. The idea would be to apply modifications to a traditional OOO GPP, either at the microarchitecture level or through dynamic translation, to enable efficient execution of nested-loop regions through selectively eschewing instruction-precise recoverability and providing explicit-dataflow execution.

Overall, in the context of dataflow research, our work has shown how traditional Von Neumann OOO and explicit-dataflow architectures favor different workload properties, and that fine-grain interleaving can provide significant and achievable benefits over either execution model alone.

Acknowledgments

We thank the anonymous reviewers, Guri Sohi, and the Vertical Research Group for their thoughtful comments. Support for this research was provided by NSF under the grant CNS-1228782 and by a Google US/Canada PhD Fellowship.

References

[1] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Trans. Comput.*, 1990.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *ISPASS*, 2005.

[4] R. Buehrer and K. Ekanadham, "Incorporating data flow ideas into von neumann processors for parallel execution," *Computers, IEEE Transactions on*, 1987.

[5] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, 2004.

[6] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *ISCA '08*.

[7] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[8] B. Fields, R. Bodik, M. Hill, and C. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *MICRO*, 2003.

[9] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley, "An evaluation of the trips computer system," in *ASPLOS '09*.

[10] D. Gibson and D. A. Wood, "Forwardflow: A scalable core for power-constrained cmps," in *ISCA*, 2010.

[11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[12] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy efficient computing," *IEEE Micro*, 2012.

[13] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White Paper*, 2011.

[14] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[15] M. Hayenga, V. Naresh, and M. Lipasti, "Revolver: Processor architecture for power efficient loop execution," in *HPCA*, 2014.

[16] C.-H. Ho, V. Govindaraju, T. Nowatzki, R. Nagaraju, Z. Marzec, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam, "Performance evaluation of a dyser fpga prototype system spanning the compiler, microarchitecture, and hardware implementation," in *ISPASS*, 2015.

[17] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *ISCA*, 2015.

[18] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *ISCA*, 1988.

[19] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture." in *ISCA*, 2004.

[20] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO*, 1997.

[21] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *ACM SIGARCH Computer Architecture News*, 2011.

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[23] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *MICRO*, 2012.

[24] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *ASPLOS*, 2006.

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[26] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Studying hybrid von-neumann/dataflow execution models," Computer Sciences Department, University of Wisconsin-Madison, Tech. Rep., 2015.

[27] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *PLDI*, 2013.

[28] S. Padmanabha, A. Lukefahr, R. Das, and S. A. Mahlke, "Trace based phase prediction for tightly-coupled heterogeneous cores," in *MICRO*, 2013.

[29] G. M. Papadopoulos, "Monsoon: an explicit token-store architecture," in *ISCA*, 1990.

[30] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability," in *MICRO*, 2012.

[31] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," in *MICRO*, 2006.

[32] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, , and C. Batten, "Architectural specialization for inter-iteration loop dependence patterns," in *MICRO*, 2014.

[33] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO*, 2003.

[34] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *ISCA*, 2014.

[35] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *ASPLOS '10*.

[36] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO*, 2011.